JAVA - STREAMS, FILES AND I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionlity related to streams and I/O. We would see most commonly used example one by one:

Reading Console Input:

Java input console is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream. Here is most common syntax to obtain BufferedReader:

```
BufferedReader br = new BufferedReader(new
                      InputStreamReader(System.in));
```

Once BufferedReader is obtained, we can use read( ) method to reach a character or readLine( ) method to read a string from the console.

Reading Characters from Console:

To read a character from a BufferedReader, we would read( ) method whose sytax is as follows:

```
int read( ) throws IOException
```

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns .1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q":

```
// Use a BufferedReader to read characters from the console.

import java.io.*;

public class BRRead {
   public static void main(String args[]) throws IOException
   {
      char c;
      // Create a BufferedReader using System.in
      BufferedReader br = new BufferedReader(new
                          InputStreamReader(System.in));
      System.out.println("Enter characters, 'q' to quit.");
      // read characters
      do {
         c = (char) br.read();
         System.out.println(c);
      } while(c != 'q');
   }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

Reading Strings from Console:

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

```
String readLine( ) throws IOException
```

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word "end":

```
// Read a string from console using a BufferedReader.
import java.io.*;
public class BRReadLines {
   public static void main(String args[]) throws IOException
   {
      // Create a BufferedReader using System.in
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      String str;
      System.out.println("Enter lines of text.");
      System.out.println("Enter 'end' to quit.");
      do {
         str = br.readLine();
         System.out.println(str);
      } while(!str.equals("end"));
   }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'end' to quit.
This is line one
This is line one
This is line two
This is line two
end
end
```

Writing Console Output:

Console output is most easily accomplished with **print( )** and **println( )**, described earlier. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

```
void write(int byteval)
```

This method writes to the stream the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written.

Example:

Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

```
import java.io.*;

// Demonstrate System.out.write().
public class WriteDemo {
   public static void main(String args[]) {
      int b;
      b = 'A';
      System.out.write(b);
      System.out.write('\n');
   }
}
```

This would produce simply 'A' character on the output screen.

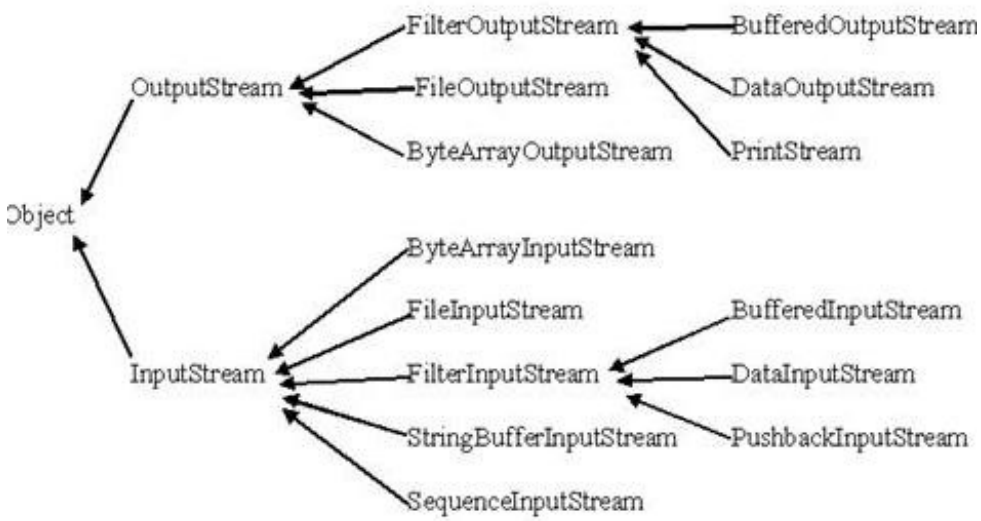
```
A
```

Note: You will not often use write( ) to perform console output because print( ) and println( ) are substantially easier to use.

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are FileInputStream and FileOutputStream which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object

using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links:

- ByteArrayInputStream
- DataInputStream

FileOutputStream:

FileOutputStream is used to create a file and write data into it.The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file.:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand then there is a list of helper methods which can be used to write to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

   try{
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("C:/test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("C:/test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java:

Creating Directories:

There are two useful **File** utility methods which can be used to create directories:

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // Create directory now.
      d.mkdirs();
  }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Reading Directories:

A directory is a File that contains a list of other files and directories. When you create a File object and it is a directory, the isDirectory( ) method will return true.

You can call list( ) on that object to extract the list of other files and directories inside. The program shown here illustrates how to use list( ) to examine the contents of a directory:

```
import java.io.File;

public class DirList {
   public static void main(String args[]) {
      String dirname = "/java";
      File f1 = new File(dirname);
      if (f1.isDirectory()) {
         System.out.println( "Directory of " + dirname);
         String s[] = f1.list();
         for (int i=0; i < s.length; i++) {
            File f = new File(dirname + "/" + s[i]);
            if (f.isDirectory()) {
               System.out.println(s[i] + " is a directory");
            } else {
               System.out.println(s[i] + " is a file");
            }
         }
      } else {
         System.out.println(dirname + " is not a directory");
```

```
    }
  }
}
```

This would produce following result:

```
Directory of /mysql
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```